

# Observing Custom Software Modifications: A Quantitative Approach of Tracking the Evolution of Patch Stacks

Ralf Ramsauer  
Technical University of Applied  
Sciences Regensburg  
ralf.ramsauer@othr.de

Daniel Lohmann  
Friedrich-Alexander University  
Erlangen-Nuremberg  
lohmann@cs.fau.de

Wolfgang Mauerer  
Technical University of Applied  
Sciences Regensburg  
Siemens AG, Munich  
wolfgang.mauerer@othr.de

## ABSTRACT

Modifications to open-source software (OSS) are often provided in the form of “patch stacks” – sets of changes (patches) that modify a given body of source code. Maintaining patch stacks over extended periods of time is problematic when the underlying base project changes frequently. This necessitates a continuous and engineering-intensive adaptation of the stack. Nonetheless, long-term maintenance is an important problem for changes that are not integrated into projects, for instance when they are controversial or only of value to a limited group of users.

We present and implement a methodology to systematically examine the temporal evolution of patch stacks, track non-functional properties like integrability and maintainability, and estimate the eventual economic and engineering effort required to successfully develop and maintain patch stacks. Our results provide a basis for quantitative research on patch stacks, including statistical analyses and other methods that lead to actionable advice on the construction and long-term maintenance of custom extensions to OSS.

## 1. INTRODUCTION

Special-purpose software, like industrial control, medical analysis, or other domain-specific applications, is often composed of contributions from general-purpose projects that provide basic building blocks. Custom modifications implemented on top of them fulfill certain additional requirements, while the development of *mainline*, the primary branch of the base project, proceeds independently.

Especially for software with high dependability requirements, it is crucial to keep up to date with mainline: latest fixes must be applied and new general features have to be introduced, as diverging software branches are hard to maintain and lead to inflexible systems [6]. Parallel development often evolves in the form of *patch stacks*: feature-granular modifications of mainline releases. Because of the dynamics exhibited by modern software projects, maintaining patch stacks can become a significant issue in terms of effort and costs.

Our toolkit **PaStA**<sup>1</sup> (**P**atch **S**tack **A**nalysis) quantitatively analyses the evolution of patch stacks by mining git [5] repositories and produces data that can serve as input for statistical analysis. It compares different releases of stacks and groups similar patches (patches that lead to similar modifications) into equivalence classes. This allows us to compare

those classes against the base project to measure integrability and influence of the patch stack on the base project. Patches that remain on the external stack across releases are classified as *invariant* and are hypothesised to reflect the maintenance cost of the whole stack. A fine grained classification of different patch types that depends on the actual modifications could function as a measure for the *invasiveness* of the stack.

In summary, we claim the following contributions:

- We provide an approach and tool for observing the evolution of patch stacks.
- We propose a language-independent semi-automatic algorithm based on string distances that is suitable for detecting similar patches on patch stacks.
- We provide a case study on Preempt-RT [10], a real-time extension of the Linux kernel that enjoys widespread use in industrial appliances for more than a decade, yet has not been integrated into standard Linux. We measure its influence on mainline and visualise the development dynamics of the stack.

## 2. APPROACH

In general, a patch stack (also known as patch set) is defined as a set of patches (commits) that are developed and maintained independently of the base project. Well-known examples include the Preempt-RT Linux realtime extension, the Linux LTSI (Long Term Support Initiative) kernel, and vendor-specific Android stacks needed to port the system to a particular hardware. In many cases, patch stacks are applied on top of individual releases of an upstream version, but they do not necessarily have to be developed in a linear way [1]. The commits of the patched version of a base project are identified as the set of commit hashes that do not occur in the mainline project.

Our analysis is based on the following assumptions:

- Mainline *upstream* development takes place in one single branch.
- Every release of the patch stack is represented by a separate branch.

The work flow of **PaStA** consists of the following steps: (1) Set up a repository containing all releases of the patch stacks. (2) Identify and group similar patches across different versions of the patch stacks. (3) Compare representatives of those groups against mainline. (4) Use statistical

<sup>1</sup><https://github.com/lfd/PaStA>

methods to draw conclusions on the development and evolution of the patch stacks.

A *commit hash* provides a unique identifier for every commit: In the following,  $U$  is the set of all commit hashes of the base project, while  $P_i$  is the set of the commit hashes of a release  $i$  of the patch stacks.  $P \equiv \bigcup_i P_i$  denotes all commit hashes on the patch stacks. Note that  $P \cap U = \emptyset$ . Let  $H \equiv P \cup U$  be the set of all commit hashes of interest. A semi-automatic classification function  $\text{comp} : P \times H \rightarrow \{\text{True}, \text{False}\}$  decides whether two patches are similar or not. A detailed description of the function  $\text{comp}$  can be found in Section 2.3.

In the implementation, PaStA mines git repositories. Without loss on generality, we focus on this particular version control system because it is widely employed in current OSS development.

## 2.1 Grouping Similar Patches

Patch stacks change as they are being aligned with the changes in base project and additionally integrate or lose functionalities. New patches are pushed on top of the stack, existing patches may be amended to follow up with API changes, or patches are dropped. Because of the rapid dynamics and growth of Open Source projects [3], a significant amount of patches must manually be ported from one release of the base project to the next. Since the base project changes over time, it is necessary to continuously adapt the details of individual patches. Those adaptations can be classified in textual and higher-order conflicts [2]. Textual conflicts can be solved by manually porting the patch to the next version. In a series of patches, patches may depend on each other, so that textual conflicts in one patch lead to follow-up conflicts in further patches. Higher-order conflicts occur when a patch obtains a new (erroneous) semantic meaning after changes in the base project diverged, despite a lack of textual conflicts. Both types are known to induce high maintenance cost [9].

Even if the semantics of patches remain invariant over time (e.g., a patch introduces identical functional modifications in subsequent revisions of the patch), their textual content can change considerably over time. To track patches with unchanged semantics over time, we introduce the classifier function  $\text{comp}$  that places similar patches into equivalence classes  $R_j$ , so that  $P = \bigcup_j R_j$ . If  $\text{comp}$  were able to track the exact semantics of patches, it would hold that  $\text{comp}(a, b) = \text{yes} \Leftrightarrow a \sim b$ . But as  $\text{comp}$  can only compare textual changes, it follows that  $\text{comp}(a, b) = \text{yes} \Rightarrow a \sim b$ . This results from the fact that two similar patches between two successive versions usually have less textual changes than the first and last occurrence of the same patch. We approximate  $P \approx \bigcup_j \hat{R}_j$ .

## 2.2 Comparing Groups Against Mainline

After grouping all patches on the stacks in equivalence classes  $R_j$ , a complete representative system  $\mathcal{R} \subseteq P$  is chosen and compared against the commits in the base project. As representative of an equivalence class, we choose the patch with the latest version.  $Q = \{(r, u) | r \in \mathcal{R}, u \in U, \text{comp}(r, u) = 1\}$  denotes the set of all patches that are found in the base project.

## 2.3 Detecting Similar Patches

To group patches into equivalence classes and find them in the base project, it is necessary to detect similar commits. Generally, a commit consists of a unique hash, a descriptive message that informally summarises the modifications, and so called *diffs* [8] that describe the actual changes of the code.

Existing work on detecting similar code fragments primarily targets on detecting code duplicates [4] or on revealing code plagiarism. Possible approaches include language-dependent lexical analysis, code fingerprinting [11], or the comparison of abstract syntax trees [7]. However, all these approaches concentrate on the comparison of code fragments and not on the comparison of *similar diffs* or commits, as required in our case.

A diff of a file consists of a sequence of *hunks* that describe the changes at a textual level. Every hunk  $h$  is introduced by a range information that determines the location of the changes within a file and contains a section heading  $h_{\text{head}}$ . Section headings display “the nearest unchanged line that precedes each hunk” [8] and are determined by a regular expression. Range information is followed by the actual changes: lines  $h^+$  that are added to the new resulting file are preceded by ‘+’, lines  $h^-$  that are removed from the original file are preceded by ‘-’ and lines  $h^\circ$  that did not change are preceded by a whitespace ‘ ’.

For the projects considered in the case study, we observed the following properties:

- Commit messages of upstream patches tend to be more verbose, but still are similar to those on patch stacks.
- Variable and identifier names do not significantly change between different versions.
- Range information of similar hunks changes between different releases.
- Section headings tend to stay similar between different releases.

In contrast to the detection of code plagiarism or the detection of code duplicates, in our case the textual content of diffs between successive releases of the patch stack tends to stay very close. For this case, string or edit distances provide an easy but powerful language independent method for detecting similar code fragments.

Comparing  $n$  diffs against each other requires  $\mathcal{O}(n^2)$  comparison operations. As the necessary string operations are computationally intensive, we employ a coarse-grained pre-evaluation that serves as a filter: Two commits can only be similar if both touch at least one common file. If the intersection of touched files is disjoint the two commits are automatically considered to be not similar.

Our algorithm calculates a rating for the similarity of the commit message and a rating for the similarity of the diff. When comparing diffs, only similar hunks of commonly changes files are compared. Insertions and deletions are compared independently.

Algorithm 1 describes the evaluation of two patches. The algorithm calculates two ratings, a message rating  $r_m \in [0, 1]$  and a diff rating  $r_d \in [0, 1]$ .  $r$  is the weighted arithmetic mean of  $r_m$  and  $r_d$ , weighted by a heuristic factor  $w \in [0, 1]$ . If the resulting rating  $r < t_i$ , the two commit hashes are

classified as dissimilar, if  $t_i \leq r < t_a$ , then manual evaluation is required, and if  $r \geq t_a$ , the commits are classified as similar. Given a commit hash, GETCOMMIT returns the corresponding message and diff. STRIPTAGS removes all *tags* (CC:, Signed-off-by:, Acked-by:, ...) as they are not relevant for comparing the content of commit messages. Given the diff of a commit, CHANGEDFILES returns all touched files of the diff. GETHUNKS returns all hunks of the diff of a file while HUNKBYHEADING searches for the closest hunk which heading matches  $x$  with a rating of at least  $t_h$  given a section heading  $x$  and the diff of a file. DIST takes either two strings or two lists of strings and returns a rating between 0 and 1, where 0 denotes no commonalities and 1 denotes absolute similarity. Our implementation uses the Levenshtein distance, which is a well-known metric of measuring the similarity of strings.

---

**Algorithm 1** Detection of similar patches

---

```

1: function COMP( $a, b, t_a, t_i, t_h, w$ )
2:   if not PREVAL( $a, b$ ) then
3:     return False
4:   ( $msg_a, diff_a$ )  $\leftarrow$  GETCOMMIT( $a$ )
5:   ( $msg_b, diff_b$ )  $\leftarrow$  GETCOMMIT( $b$ )
6:    $r_m \leftarrow$  DIST(STRIPTAGS( $msg_a$ ), STRIPTAGS( $msg_b$ ))
7:    $r_d \leftarrow []$ 
8:   for each file  $\leftarrow$  CHANGEDFILES( $diff_a$ ) do
9:     hunks $_a \leftarrow$  GETHUNKS( $diff_a$ , file)
10:    hunks $_b \leftarrow$  GETHUNKS( $diff_b$ , file)
11:     $r_f \leftarrow []$ 
12:    for each lhunk  $\leftarrow$  hunks $_a$  do
13:      rhunk  $\leftarrow$  HUNKBYHEADING(hunks $_b$ , lhunk $_{head}$ ,  $t_h$ )
14:      if rhunk is None then
15:        continue
16:       $r_f.append(DIST(lhunk^+, rhunk^+))$ 
17:       $r_f.append(DIST(lhunk^-, rhunk^-))$ 
18:     $r_d.append(MEAN(r_f))$ 
19:   $r_d \leftarrow MEAN(r_d)$ 
20:   $r \leftarrow w \cdot r_m + (1 - w) \cdot r_d$ 
21:  if  $r \geq t_a$  then
22:    return True
23:  else if  $r \geq t_i$  then
24:    return INTERACTIVEREVIEW( $a, b$ )
25:  return False

```

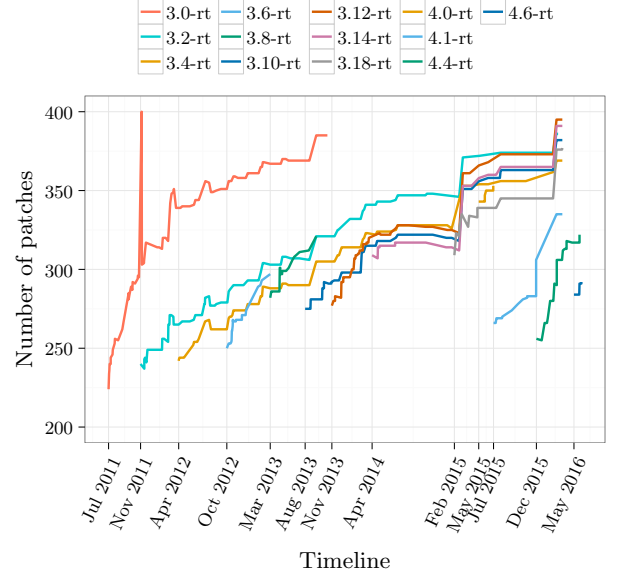
---

### 3. DISCUSSION

After grouping all patches into equivalence classes and linking them to optional commits of the base project, we can distinguish between two temporal conditions: (1) Patches that first appeared on the patch stack and later appeared in the base project (ports or *forwardports*) and (2) patches that first appeared in the base project and were ported back to older versions of the stack (backports). Patches that are not linked to a commit of the base project are called *invariant*, as they only appear on the stack.

Across two releases of the patch stack, we observe a flow of patches: (1) inflow – new patches on the patch stack and backports. (2) outflow – patches that went upstream or patches that were dropped. (3) invariant – patches that remain on the stack.

In the following, we consider the evolution of the Preempt-RT patch stack as a case study: First, we inspect the tem-



**Figure 1: Preempt-RT patch stack: Evolution of the stack size since Linux kernel version 3.0**

poral evolution of patch stack size, which is visualised in Figure 1. Among all 554 releases of the patch stack published since July 2011 (that in total consist of almost 173 000 patches), we detected 1042 different groups of patches. 195 of those groups were classified as backports, 153 groups were classified as forwardports.

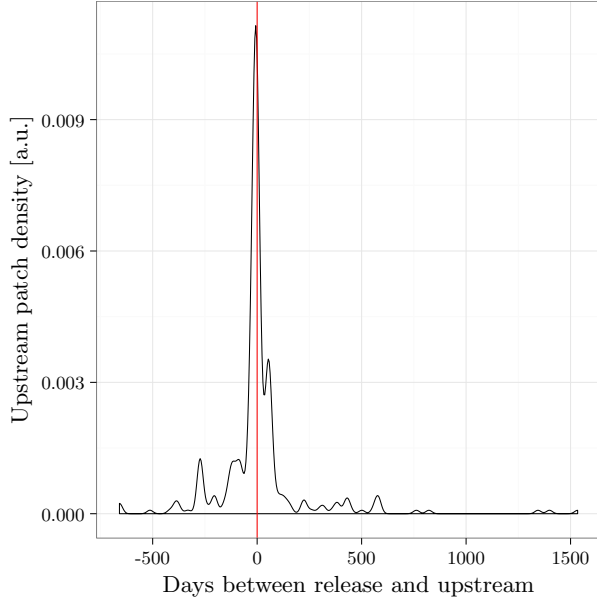
Knowledge of the stack history allows us to determine the composition of older patch stacks in terms of the direction of flow of constituents. Retroactively, we can determine which patches of the stack went upstream at a later point in time, and compute the amount of backported patches and invariant patches. Figure 3 shows the composition of the latest releases of major versions of the Preempt-RT[10] patch stack. Green bars describe the amount of patches on the stack that eventually are integrated into the upstream code base, red bars describe the amount of backports, and the blue bars give the number of invariant patches.

Another covariate of interest is the duration a patch needs to go upstream (i.e., the time between the first appearance on the patch stack and the integration with the base project). Figure 2 shows the result of this analysis for the Preempt-RT project. Positive values on the  $x$ -axis describe forwardports, negative values describe backports. There is a prominent hot spot around zero days. We interpret this spot to indicate close cooperation with the base project: backporting of many patches only takes few days while the author list of forward and backport patches overlaps.

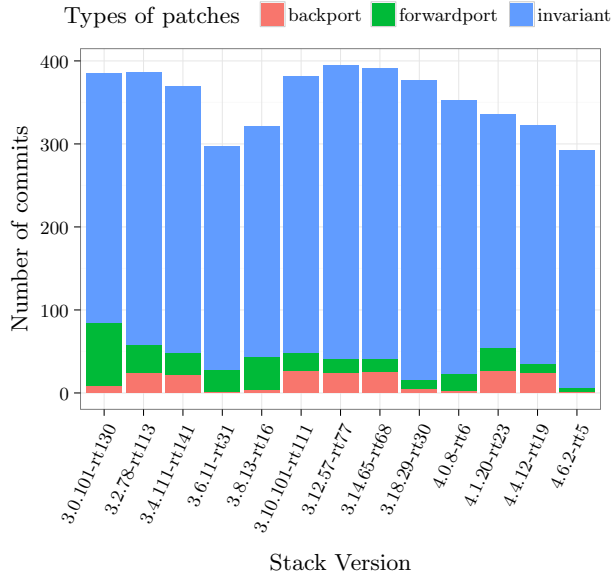
### 4. CONCLUSIONS

We presented an approach and implementation for the quantitative analysis of patch stacks and a semi-automatic method for identifying similar commits. An evaluation and visualisation of the Preempt-RT patch stack was presented as case study.

In future work, we will concentrate on deeper statistical analysis and comparing the properties and software-engineer-



**Figure 2: Preempt-RT patch stack: Distribution of integration times (in days) for patches that are eventually integrated in mainline. Positive values indicate forwardports, negative values indicate backports.**



**Figure 3: Preempt-RT patch stack: Comparing the composition of the last major releases of the patch stacks**

ring implications of patch stacks for a various projects. We are also working on a measure to quantify the invasiveness of patches and patch stacks, which will allow us to draw conclusions on the eventual maintenance cost of such stacks.

## 5. REFERENCES

- [1] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 1–10, May 2009.
- [2] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11*, pages 168–178, New York, NY, USA, 2011. ACM.
- [3] A. Deshpande and D. Riehle. *Open Source Development, Communities and Quality: IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software, September 7-10, 2008, Milano, Italy*, chapter The Total Growth of Open Source, pages 197–209. Springer US, Boston, MA, 2008.
- [4] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE, 1999.
- [5] git version control system. <https://git-scm.com/>.
- [6] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 342–352, Piscataway, NJ, USA, 2012. IEEE Press.
- [7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.
- [8] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files*, 2013. <http://www.gnu.org/software/diffutils/manual/diffutils.pdf>.
- [9] H. Munakata and T. Shibata. *The Economic Value of the Long-Term Support Initiative (LTSI)*. Linux Foundation, 2013.
- [10] Preempt-RT Wiki. <https://rt.wiki.kernel.org/>.
- [11] R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *Proceedings of the International Workshop on Software Clones (IWSC)*, 2009.